Dynamic Provenance for SPARQL Updates Using Named Graphs

Dynamic Provenance for SPARQL Updates

Harry Halpin and James Cheney

[1] World Wide Web Consortium/MIT, 32 Vassar Street Cambridge, MA 02139 USA
hhalpin@w3.org,
[2] University of Edinburgh, School of Informatics,
10 Crichton St.,
Edinburgh UK EH8 9AB

Abstract. While the Semantic Web currently can exhibit provenance information by using the W3C PROV standards, there is a "missing link" in connecting PROV to storing and querying for dynamic changes to RDF graphs using SPARQL. Solving this problem would be required for such clear use-cases as the creation of version control systems for RDF. While some provenance models and annotation techniques for storing and querying provenance data originally developed with databases or workflows in mind transfer readily to RDF and SPARQL, these techniques do not readily adapt to describing changes in dynamic RDF datasets over time. In this paper we explore how to adapt the dynamic copy-paste provenance model of Buneman et al. [2] to RDF datasets that change over time in response to SPARQL updates, how to represent the resulting provenance records themselves as RDF in a manner compatible with W3C PROV, and how the provenance information can be defined by reinterpreting SPARQL updates. The primary contribution of this paper is a semantic framework that enables the semantics of SPARQL Update to be used as the basis for a 'cut-and-paste' provenance model in a principled manner.

Keywords: SPARQL Update, provenance, versioning, RDF, semantics

1 Introduction

It is becoming increasingly common to publish scientific and governmental data on the Web as RDF (the Resource Description Framework, a W3C standard for structured data on the Web) and to attach provenance data to this information using the W3C PROV standard. In doing so, it is crucial to track not only the provenance metadata, but the changes to the graph itself, including both its derivation process and a history of changes to the data over time. Being able to track changes to RDF graphs could, in combination with the W3C PROV standards and SPARQL, provide the foundation for addressing the important use-case of creating a Git-like version control system for RDF.

The term provenance is used in several different ways, often aligning with research in two different communities, in particular the database community and the scientific workflow community. We introduce some terminology to distinguish different uses of the term. There is a difference between *static* provenance that describes data at a given point in time versus *dynamic* provenance that describes how artifacts have evolved over time. Second, there is a difference between provenance for *atomic* artifacts that expose

no internal structure as part of the provenance record, versus provenance for *collections* or other structured artifacts. The workflow community has largely focused on static provenance for atomic artifacts, whereas much of the work on provenance in databases has focused on dynamic provenance for collections (e.g. tuples in relations). An example of static provenance would be attaching the name of an origin and time-stamp to a set of astronomical RDF data. Thus static provenance can often be considered metadata related to provenance. Currently the W3C PROV data model and vocabulary [15], provides a standard way to attach this provenance, and other work such as PROV-AQ [13] provides options for extracting this metadata by virtue of HTTP.

An example of *dynamic provenance* would be given by a single astronomical data-set that is updated over time, and then if some erroneous data was added at a particular time, the data-set could be queried for its state at a prior time before the erroneous data was added so the error could be corrected. Thus dynamic provenance can in some cases be reduced to issues of history and version control. Note that static and dynamic provenance are not orthogonal, as we can capture *static* provenance metadata at every step of recording dynamic provenance. The W3C Provenance Working Group has focused primarily on static provenance, but we believe a mutually beneficial relationship between the W3C PROV static provenance and SPARQL Update with an improved provenance-aware semantic model will allow dynamic provenance capabilities to be added to RDF. While fine-grained dynamic provenance imposes overhead that may make it unsuited to some applications, there are use-cases where knowing exactly when a graph was modified is necessary for reasons of accountability, including data-sets such as private financial and public scientific data.

1.1 Related Literature

The workflow community has largely focused on declaratively describing causality or derivation steps of processes to aid repeatability for scientific experiments, and these requirements have been a key motivation for the Open Provenance Model (OPM) [17,18], a vocabulary and data model for describing processes including (but certainly not limited to) runs of scientific workflows. OPM is important in its own right, and has become the foundation for the W3C PROV data model [19]. The formal semantics of PROV have been formalized, but do not address the relationship between the provenance and the semantics of the processes being described [7]. However, previous work on the semantics of OPM, PROV, and other provenance vocabularies focuses on temporal and validity constraints [14] and does not address the meaning of the processes being represented — one could in principle use them either to represent static provenance for processes that construct new data from existing artifacts or to represent dynamic provenance that represents how data change over time at each step, such as needed by version control systems. Most applications of OPM seem to have focused on static provenance, although PROV explicitly grapples with issues concerning representing the provenance of objects that may be changing over time, but does not provide a semantics for storing and querying changes over time. Our approach to dynamic provenance is complementary to work on OPM and PROV, as we show how any provenance metadata vocabulary such as the W3C PROV ontology [15] can be connected directly to query and update

languages by presenting a semantics for a generalized provenance model and showing how this builds upon the formal semantics of the query and update languages.

Within database research, complex provenance is also becoming increasingly seen as necessary, although the work has taken a different turn than that of the workflow community. Foundational work on provenance for database queries distinguishes between *where-provenance*, which is the "locations in the source databases from which the data was extracted," and *why-provenance*, which is "the source data that had some influence on the existence of the data" [4]. Further, increasing importance is being placed on *how-provenance*, the operations used to produce the derived data and other annotations, such as who precisely produced the derived data and for what reasons. There is less work considering provenance for updates; previous work on provenance in databases has focused on simple atomic update operations: insertion, deletion, and copy [2]. A provenance-aware database should support queries that permit users to find the ultimate or proximate 'sources' of some data and apply data provenance techniques to understand why a given part of the data was inserted or deleted by a given update. There is considerable work on correct formalisms to enable provenance in databases [9]. This work on dynamic provenance is related to earlier work on version control in unstructured data, a classic and well-studied problem for information systems ranging from source code management systems to temporal databases and data archiving [23]. Version control systems such as CVS, Subversion, and Git have a solid track record of tracking versions and (coarse-grained) provenance for text (e.g. source code) over time and are well-understood in the form of temporal annotations and a log of changes.

On the Semantic Web, work on provenance has been quite diverse. There is widely implemented support for *named graphs*, where each graph G is identified with a *name URI* [6], and as the new RDF Working Group has standardized the graph name in the semantics, the new standards have left the practical use of such a graph name underdefined; thus, the graph name could be used to store or denote provenance-related information such as time-stamps. Tackling directly the version control aspect of provenance are proposals such as Temporal RDF [10] for attaching temporal annotations and a generalized Annotated RDF for any partially-ordered sets already exist [25,16]. However, by confining provenance annotations to partially-ordered sets, these approaches do not allow for a (queryable) graph structure for more complex types of provenance that include work such as the PROV model. Static provenance techniques have been investigated for RDFS inferences [5,8,20] over RDF datasets. Some of this work considers updates, particularly Flouris et al. [8], who consider the problem of how to maintain provenance information for RDFS inferences when tuples are inserted or deleted using coherence semantics. Their solution uses 'colouring' (where the color is the URI of the source of each triple) and tracking implicit triples [8].

Understanding provenance for a language requires understanding the ordinary semantics of the language. Arenas et al. formalized the semantics of SPARQL [21,1], and the SPARQL Update recommendation proposes a formal model for updates [22]. Horne et al. [12] propose an operational semantics for SPARQL Updates, which however differs in some respects from the SPARQL 1.1 standard and does not deal with named graphs.

1.2 Overview

In this paper, we build on Arenas et al.'s semantics of SPARQL [21,1] and extend it to formalize SPARQL Update semantics (following a denotational approach similar to the SPARQL Update Formal Model [22]). Then, as our main contribution, we detail a provenance model for SPARQL queries and updates that provides a (queryable) record of how the raw data in a dataset has changed over time. This change history includes a way to insert static provenance metadata using a full-scale provenance ontology such as PROV-O [15].

Our hypothesis is that a simple vocabulary, composed of insert, delete, and copy operations as introduced by Buneman et al. [2], along with explicit identifiers for update steps, versioning relationships, and metadata about updates provides a flexible format for dynamic provenance on the Semantic Web. A primary advantage of our methodology is it keeps the changes to raw data separate from the changes in provenance metadata, so legacy applications will continue to work and the cost of storing and providing access to provenance can be isolated from that of the raw data. We will introduce the semantics of SPARQL queries, then our semantics for SPARQL updates, and finally describe our semantics for dynamic provenance-tracking for RDF graphs and SPARQL updates. To summarize, our contributions are an extension to the semantics of SPARQL Update that includes provenance semantics to handle dynamic semantics [21,1] and a vocabulary for representing changes to RDF graphs made by SPARQL updates, and a translation from ordinary SPARQL updates to provenance-aware updates that record provenance as they execute.

2 Background: Semantics of SPARQL Queries

We first review a simplified (due to space limits) version of Arenas et. al.'s formal semantics of SPARQL [21,1]. Note that this is not original work, but simply a necessary precursor to the semantics of SPARQL Update and our extension that adds provenance to SPARQL Update. The main simplification is that we disallow nesting other operations such as `.`, `UNION`, etc. inside `GRAPH` $A\,\{\ldots\}$ patterns. This limitation is inessential.

Let Lit be a set of literals (e.g. strings), let Id be a set of resource identifiers, and let Var be a set of variables usually written $?X$. We write $\mathsf{Atom} = \mathsf{Lit} \cup \mathsf{Id}$ for the set of atomic values, that is literals or ids. The syntax of a core algebra for SPARQL discussed in [1] is as follows:

$$
\begin{aligned}
A &::= \ell \in \mathsf{Lit} \mid \iota \in \mathsf{Id} \mid ?X \in \mathsf{Var} \\
t &::= \langle A_1\ A_2\ A_3 \rangle \\
C &::= \{t_1, \ldots, t_n\} \mid \texttt{GRAPH}\ A\ \{t_1, \ldots, t_n\} \mid C\ C' \\
R &::= \texttt{BOUND}(?x) \mid A = B \mid R \wedge R' \mid R \vee R' \mid \neg R \\
P &::= C \mid P\ \texttt{.}\ P' \mid P\ \texttt{UNION}\ P' \mid P\ \texttt{OPT}\ P' \mid P\ \texttt{FILTER}\ R \\
Q &::= \texttt{SELECT}\ ?\overline{X}\ \texttt{WHERE}\ P \mid \texttt{CONSTRUCT}\ C\ \texttt{WHERE}\ P
\end{aligned}
$$

Here, C denotes basic graph (or dataset) patterns that may contain variables; R denotes conditions; P denotes patterns, and Q denotes queries. We do not distinguish

between subject, predicate and object components of triples, so this is a mild generalization of [1], since SPARQL does not permit literals to appear in the subject or predicate position or as the name of a graph in the $\texttt{GRAPH}\ A\ \{P\}$ pattern, although the formal semantics of RDF allows this and the syntax may be updated in forthcoming work on RDF. We also do not consider blank nodes, which pose complications especially when updates are concerned, and we instead consider them to be skolemized (or just replaced by generic identifiers), as this is how most implementations handle blank nodes. There has been previous work giving a detailed treatment of the problematic nature of blank nodes and why skolemization is necessary in real-world work.

The semantics of queries Q or patterns P is defined using functions from *graph stores* $\mathcal{D}$ to sets of *valuations* μ. A graph store $\mathcal{D} = (G, \{g_i \mapsto G_1 \ldots, g_n \mapsto G_n\})$ consists of a default graph G_0 together with a mapping from names g_i to graphs G_i. Each such graph is essentially just a set of ground triples. We often refer to graph stores as *datasets*, although this is a slight abuse of terminology.

We overload set operations for datasets, e.g. $\mathcal{D} \cup \mathcal{D}'$ or $\mathcal{D} \setminus \mathcal{D}'$ denotes the dataset obtained by unioning or respectively subtracting the default graphs and named graphs of $\mathcal{D}$ and $\mathcal{D}'$ pointwise. If a graph g is defined in $\mathcal{D}$ and undefined in $\mathcal{D}'$, then $(\mathcal{D} \cup \mathcal{D}')(g) = \mathcal{D}(g)$ and similarly if g is undefined in $\mathcal{D}$ and defined in $\mathcal{D}'$ then $(\mathcal{D} \cup \mathcal{D}')(g) = \mathcal{D}'(g)$; if g is undefined in both datasets then it is undefined in their union. For set difference, if g is defined in $\mathcal{D}$ and undefined in $\mathcal{D}'$ then $(\mathcal{D} \setminus \mathcal{D}')(g) = \mathcal{D}(g)$; if g is undefined in $\mathcal{D}$ then it is undefined in $(\mathcal{D} \setminus \mathcal{D}')$. Likewise, we define $\mathcal{D} \subseteq \mathcal{D}'$ as $\mathcal{D}' = \mathcal{D} \cup \mathcal{D}'$.

A *valuation* is a partial function $\mu : \mathsf{Var} \rightharpoonup \mathsf{Lit} \cup \mathsf{Id}$. We lift valuations to functions $\mu : \mathsf{Atom} \cup \mathsf{Var} \to \mathsf{Atom}$ as follows:

$$\begin{aligned} \mu(?X) &= \mu(X) \\ \mu(a) &= a \qquad a \in \mathsf{Atom} \end{aligned}$$

that is, if A is a variable $?X$ then $\mu(A) = \mu(X)$ and otherwise if A is an atom then $\mu(A) = A$. We thus consider all atoms to be implicitly part of the domain of μ. Furthermore, we define μ applied to triple, graph or dataset patterns as follows:

$$\begin{aligned} \mu(\langle A_1\ A_2\ A_3 \rangle) &= \langle \mu(A_1)\ \mu(A_2)\ \mu(A_3) \rangle \\ \mu(\{t_1, \ldots, t_n\}) &= (\{\mu(t_1), \ldots, \mu(t_n)\}, \emptyset) \\ \mu(\texttt{GRAPH}\ A\ \{t_1, \ldots, t_n\}) &= (\emptyset, \{\mu(A) \mapsto \{\mu(t_1), \ldots, \mu(t_n)\}\}) \\ \mu(C\ C') &= \mu(C) \cup \mu(C') \end{aligned}$$

where, as elsewhere, we define $\mathcal{D} \cup \mathcal{D}'$ as pointwise union of datasets.

The conditions R are interpreted as three-valued formulas over the lattice $L = \{\texttt{true}, \texttt{false}, \texttt{error}\}$, where $\texttt{false} \leq \texttt{error} \leq \texttt{true}$, and $\wedge$ and $\vee$ are minimum and maximum operations respectively, and $\neg\texttt{true} = \texttt{false}$, $\neg\texttt{false} = \texttt{true}$, and

$\neg\texttt{error} = \texttt{error}$. The semantics of a condition is defined as follows:

$$[\![\texttt{BOUND}(?X)]\!]\mu = \begin{cases} \texttt{false} \text{ if } ?X \notin \mathrm{dom}(\mu) \\ \texttt{true} \quad \text{if } ?X \in \mathrm{dom}(\mu) \end{cases}$$

$$[\![A = B]\!]\mu = \begin{cases} \texttt{error} \text{ if } \{A, B\} \not\subseteq \mathrm{dom}(\mu) \\ \texttt{true} \quad \text{if } \mu(A) = \mu(B) \text{ where } A, B \in \mathrm{dom}(\mu) \\ \texttt{false} \text{ if } \mu(A) \neq \mu(B) \text{ where } A, B \in \mathrm{dom}(\mu) \end{cases}$$

$$[\![\neg R]\!]\mu = \neg[\![R]\!]\mu$$

$$[\![R \wedge R']\!]\mu = [\![R]\!]\mu \wedge [\![R']\!]\mu$$

$$[\![R \vee R']\!]\mu = [\![R]\!]\mu \vee [\![R']\!]\mu$$

We write $\mu \models R$ to indicate that $[\![R]\!] = \texttt{true}$.

We say that two valuations μ, μ' are *compatible* (or write μ compat μ') if for all variables $x \in \mathrm{dom}(\mu) \cap \mathrm{dom}(\mu')$, we have $\mu(x) = \mu'(x)$. Then there is a unique valuation $\mu \cup \mu'$ that behaves like μ on $\mathrm{dom}(\mu)$ and like μ' on $\mathrm{dom}(\mu')$. We define the following operations on sets of valuations Ω.

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \text{ compat } \mu_2\}$$

$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$$

$$\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \mid \nexists \mu' \in \Omega_2 . \mu \text{ compat } \mu'\}$$

$$\Omega_1 \ltimes \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$$

Note that union is the same as ordinary set union, but difference is not, since $\Omega_1 \setminus \Omega_2$ only includes valuations that are incompatible with all those in in Ω_2.

Now we can define the meaning of a pattern P in a dataset $\mathcal{D}$ as a set of valuations $[\![P]\!]_{\mathcal{D}}$, as follows:

$$[\![C]\!]_{\mathcal{D}} = \{\mu \mid \mathrm{dom}(\mu) = vars(C) \text{ and } \mu(C) \subseteq \mathcal{D}\}$$

$$[\![P_1 \ . \ P_2]\!]_{\mathcal{D}} = [\![P_1]\!]_{\mathcal{D}} \bowtie [\![P_2]\!]_{\mathcal{D}}$$

$$[\![P_1 \ \texttt{UNION} \ P_2]\!]_{\mathcal{D}} = [\![P_1]\!]_{\mathcal{D}} \cup [\![P_2]\!]_{\mathcal{D}}$$

$$[\![P_1 \ \texttt{OPT} \ P_2]\!]_{\mathcal{D}} = [\![P_1]\!]_{\mathcal{D}} \ltimes [\![P_2]\!]_{\mathcal{D}}$$

$$[\![P \ \texttt{FILTER} \ R]\!]_{\mathcal{D}} = \{\mu \in [\![P]\!]_{\mathcal{D}} \mid \mu \models R\}$$

Note that, in contrast to Arenas et al.'s semantics for SPARQL with named graphs [1], we do not handle `GRAPH` A $\{\ldots\}$ patterns that contain other pattern operations such as `.` or `UNION`, and we do not keep track of the "current graph" G. Instead, since graph patterns can only occur in basic patterns, we can build the proper behavior of pattern matching into the definition of $\mu(C)$, and we select all matches μ such that $\mu(C) \subseteq \mathcal{D}$ in the case for $[\![C]\!]_{\mathcal{D}}$.

Finally, we consider the semantics of selection and construction queries. A selection query has the form `SELECT` $?\overline{X}$ `WHERE` P where $?\overline{X}$ is a list of distinct variables. It simply returns the valuations obtained by P and discards the bindings of variables not in $\overline{X}$. A construction query builds a new graph or dataset from these results. Note that in SPARQL such queries only construct anonymous graphs; here we generalize in order

to use construction queries to build datasets that can be inserted or deleted.

$$[\![\texttt{SELECT}\ ?\overline{X}\ \texttt{WHERE}\ P]\!]_{\mathcal{D}} = \{\mu|_{\overline{X}} \mid \mu \in [\![P]\!]_{\mathcal{D}}\}$$
$$[\![\texttt{CONSTRUCT}\ C\ \texttt{WHERE}\ P]\!]_{\mathcal{D}} = \bigcup\{\mu(C) \mid \mu \in [\![P]\!]_{\mathcal{D}}\}$$

Here, note that $\mu|_{\overline{X}}$ stands for μ restricted to the variables in the list $\overline{X}$.

We omit discussion of the FROM components of queries (which are used to initialize the graph store by pulling data in from external sources) or of the other query forms ASK, and DESCRIBE, as they are described elsewhere [1] in a manner coherent with our approach.

3 The Semantics of SPARQL Update

We will describe the semantics of the core language for atomic updates, based upon [22]:

$$\begin{aligned} U ::= {} & \texttt{INSERT}\ \{C\}\ \texttt{WHERE}\ P \mid \texttt{DELETE}\ \{C\}\ \texttt{WHERE}\ P \\ \mid {} & \texttt{DELETE}\ \{C\}\ \texttt{INSERT}\ \{C'\}\ \texttt{WHERE}\ P \mid \texttt{LOAD}\ g\ \texttt{INTO}\ g' \mid \texttt{CLEAR}\ g \\ \mid {} & \texttt{CREATE}\ g \mid \texttt{DROP}\ g \mid \texttt{COPY}\ g\ \texttt{TO}\ g' \mid \texttt{MOVE}\ g\ \texttt{TO}\ g' \mid \texttt{ADD}\ g\ \texttt{TO}\ g' \end{aligned}$$

We omit the INSERT DATA and DELETE DATA forms since they are definable in terms of INSERT and DELETE.

SPARQL Update [22] specifies that transactions consisting of multiple updates should be applied atomically, but leaves some semantic questions unresolved, such as whether aborted transactions have to roll-back partial changes. It also does not specify whether updates in a transaction are applied sequentially (as in most imperative languages), or using a snapshot semantics (as in most database update languages). Both alternatives pose complications, so in this paper we focus on transactions consisting of single atomic updates.

We model a collection of named graphs as a dataset $\mathcal{D}$, as for SPARQL queries. We consider only a single graph in isolation here, and not the case of multiple named graphs that may be being updated concurrently. The semantics of an update operation u on dataset $\mathcal{D}$ is defined as $[\![U]\!]_{\mathcal{D}}$.

The semantics of a SPARQL Update U in dataset $\mathcal{D}$ is defined as follows:

$$
\begin{aligned}
[\![\texttt{DELETE}\ \{C\}\ \texttt{WHERE}\ P]\!]_{\mathcal{D}} &= \mathcal{D} \setminus [\![\texttt{CONSTRUCT}\ C\ \texttt{WHERE}\ P]\!]_{\mathcal{D}} \\
[\![\texttt{INSERT}\ \{C\}\ \texttt{WHERE}\ P]\!]_{\mathcal{D}} &= \mathcal{D} \cup [\![\texttt{CONSTRUCT}\ C\ \texttt{WHERE}\ P]\!]_{\mathcal{D}} \\
[\![\texttt{DELETE}\ \{C\}\ \texttt{INSERT}\ \{C'\}\ \texttt{WHERE}\ P]\!]_{\mathcal{D}} &= (\mathcal{D} \setminus [\![\texttt{CONSTRUCT}\ C\ \texttt{WHERE}\ P]\!]_{\mathcal{D}}) \\
&\quad \cup [\![\texttt{CONSTRUCT}\ C'\ \texttt{WHERE}\ P]\!]_{\mathcal{D}} \\
[\![\texttt{LOAD}\ g_1\ \texttt{INTO}\ g_2]\!]_{\mathcal{D}} &= \mathcal{D}[g_2 := \mathcal{D}(g_1) \cup \mathcal{D}(g_2)] \\
[\![\texttt{CLEAR}\ g]\!]_{\mathcal{D}} &= \mathcal{D}[g := \emptyset] \\
[\![\texttt{CREATE}\ g]\!]_{\mathcal{D}} &= \mathcal{D} \uplus \{g \mapsto \emptyset\} \\
[\![\texttt{DROP}\ g]\!]_{\mathcal{D}} &= \mathcal{D}[g := \bot] \\
[\![\texttt{COPY}\ g_1\ \texttt{TO}\ g_2]\!]_{\mathcal{D}} &= \begin{cases} \mathcal{D}[g_2 := \mathcal{D}(g_1)] & \text{if } g_1 \neq g_2 \\ \mathcal{D} & \text{otherwise} \end{cases} \\
[\![\texttt{MOVE}\ g_1\ \texttt{TO}\ g_2]\!]_{\mathcal{D}} &= \begin{cases} \mathcal{D}[g_2 := \mathcal{D}(g_1), g_1 := \bot] & \text{if } g_1 \neq g_2 \\ \mathcal{D} & \text{otherwise} \end{cases} \\
[\![\texttt{ADD}\ g_1\ \texttt{TO}\ g_2]\!]_{\mathcal{D}} &= \mathcal{D}[g_2 := \mathcal{D}(g_1) \cup \mathcal{D}(g_2)]
\end{aligned}
$$

Here, $\mathcal{D}[g := G]$ denotes $\mathcal{D}$ updated by setting the graph named g to G, and $\mathcal{D}[g := \bot]$ denotes $\mathcal{D}$ updated by making g undefined, and finally $\mathcal{D} \uplus [g := G]$ denotes $\mathcal{D}$ updated by adding a graph named g with value G, where g must not already be in the domain of $\mathcal{D}$. Set-theoretic notation is used for graphs, e.g. $G \cup G'$ is used for set union and $G \setminus G'$ for set difference, and $\emptyset$ stands for the empty graph. Note that the `COPY` g `TO` g' and `MOVE` g `TO` g' operations have no effect if $g = g'$. Also, observe that we do not model external URI dereferences, and the `LOAD` g `INTO` g' operation (which allows g to be an external URI) behaves exactly as `ADD` g `TO` g' operation (which expects g to be a local graph name).

4 Provenance semantics

A single SPARQL update can read from and write to several named graphs (and possibly also the default graph). For simplicity, we restrict attention to the problem of tracking the provenance of updates to a single (possibly named) RDF graph. All operations may still use the default graph or other named graphs in the dataset as sources. The general case can be handled using the same ideas as for a single anonymous graph, only with more bureaucracy to account for versioning of all of the named graphs managed in a given dataset.

A graph that records all the updates of triples from a given graph g is considered a *provenance graph* for g. For each operation, a *provenance record* is stored that track of the state of the graph at any given moment and their associated metadata. The general concept is that in a fully automated process one should be able to re-construct the state of the given graph at any time from its provenance graph by following the SPARQL queries and metadata given in the provenance records for each update operation tracked.

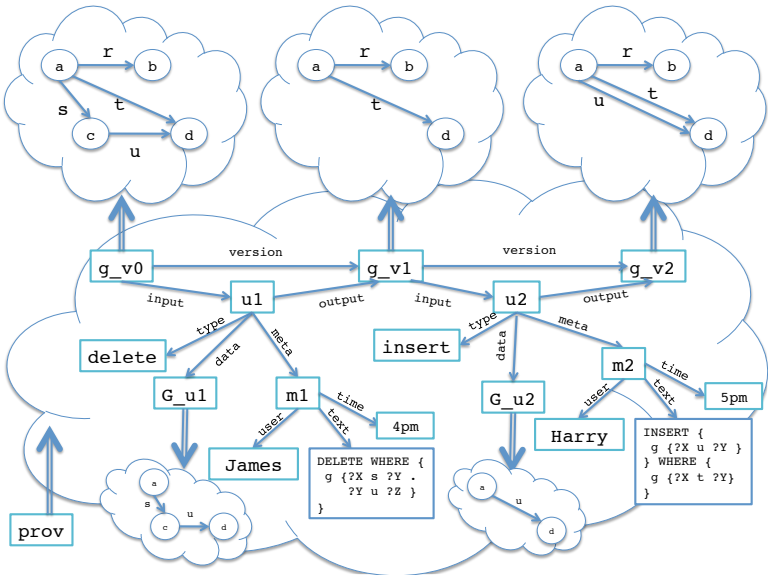

Fig. 1. Example provenance graph

We model the provenance of a single RDF graph G that is updated over time as a set of history records, including the special provenance graph named $prov$ which consists of indexed graphs for each operation such as `G_v0`,...,`G_vn` and `G_u1`...,`G_um`. These provenance records are immutable; that is, once they have been created and initialized, the implementation should fixed so that their content cannot be changed. The index of all provenance records then is also strictly linear and consistent (i.e. non-circular), although branching could be allowed. They can be stored as a special immutable type in the triple-store. Intuitively, `G_vi` is the named graph showing G's state in version i and `G_ui` is another named graph showing the triples inserted into or deleted from G by update i. An example illustration is given in Figure 1.

The provenance graph of named graph `G` includes several kinds of nodes and edges:

- `G_vi upd:version G_vi+1` edges that show the sequence of versions. Whenever a `upd:version` link is added between `G_vi` and `G_vi+1`, a backlink called `upd:prevVersion` between `G_vi+1` and `G_vi`;
- nodes `u1`,...,`un` representing the updates that have been applied to G, along with a `upd:type` edge linking to one of `upd:insert`, `upd:delete`, `upd:load`, `upd:clear`, `upd:create`, or `upd:drop`.
- For all updates except create, an `upd:input` edge linking `ui` to `G_vi`.
- For all updates except drop, an `upd:output` edge linking `ui` to `G_vi+1`.
- For insert and delete updates, an edge `ui upd:data G_ui` where `G_ui` is a named graph containing the triples that were inserted or deleted by `ui`.

– Edges `ui upd:source n` linking each update to each named graph `n` that was consulted by `ui`. For an insert or delete, this includes all graphs that were consulted while evaluating P (note that this may only be known at run time); for a load update, this is the name of the graph whose contents were loaded; create, drop and clear updates have no sources.
– Additional edges from `ui` providing metadata for the update (such as author, commit time, log message, or the source text of the update); possibly using a standard vocabulary such as Dublin Core, or using OPM or PROV vocabulary terms.

Note that this representation does not directly link triples in a given version to places from which they were "copied" as it only contains the triples directly concerning the update in the history record. However, each history record in combination with the rest of the history records in the provenance graph does provide enough information to recover previous versions on request. As we store the source text of the update statements performed by each update in each history record of the provenance graph, we can trace backwards through the update sequence to to identify places where triples were inserted or copied into or deleted from the graph. For queries, we consider a simple form of provenance which calculates a set of named graphs "consulted" by the query. The set of sources of a pattern or query is computed as follows:

$$\begin{aligned}
\mathcal{S}[\![C]\!]_{\mathcal{D}} &= \bigcup\{\mathrm{names}_{\mu}(C) \mid \mu \in [\![C]\!]_{\mathcal{D}}\} \\
\mathcal{S}[\![P_1 \ . \ P_2]\!]_{\mathcal{D}} &= \mathcal{S}[\![P_1]\!]_{\mathcal{D}} \cup \mathcal{S}[\![P_2]\!]_{\mathcal{D}} \\
\mathcal{S}[\![P_1 \ \texttt{UNION} \ P_2]\!]_{\mathcal{D}} &= \mathcal{S}[\![P_1]\!]_{\mathcal{D}} \cup \mathcal{S}[\![P_2]\!]_{\mathcal{D}} \\
\mathcal{S}[\![P_1 \ \texttt{OPT} \ P_2]\!]_{\mathcal{D}} &= \mathcal{S}[\![P_1]\!]_{\mathcal{D}} \cup \mathcal{S}[\![P_2]\!]_{\mathcal{D}} \\
\mathcal{S}[\![P \ \texttt{FILTER} \ R]\!]_{\mathcal{D}} &= \mathcal{S}[\![P]\!]_{\mathcal{D}} \\
\mathcal{S}[\![\texttt{SELECT} \ ?\overline{X} \ \texttt{WHERE} \ P]\!]_{\mathcal{D}} &= \mathcal{S}[\![P]\!]_{\mathcal{D}} \\
\mathcal{S}[\![\texttt{CONSTRUCT} \ C \ \texttt{WHERE} \ P]\!]_{\mathcal{D}} &= \mathcal{S}[\![P]\!]_{\mathcal{D}}
\end{aligned}$$

where the auxiliary function $\mathrm{names}_{\mu}(C)$ collects all of the graph names occurring in a ground basic graph pattern C:

$$\begin{aligned}
\mathrm{names}_{\mu}(\{t_1, \ldots, t_n\}) &= \{\texttt{DEFAULT}\} \\
\mathrm{names}_{\mu}(\texttt{GRAPH} \ A \ \{t_1, \ldots, t_n\}) &= \{\mu(A)\} \\
\mathrm{names}_{\mu}(C \ C') &= \mathrm{names}_{\mu}(C) \cup \mathrm{names}_{\mu}(C')
\end{aligned}$$

Here, we use the special identifier `DEFAULT` as the name of the default graph; this can be replaced by its URI.

The $\mathcal{S}[\![P]\!]_{\mathcal{D}}$ function produces a set S of graph identifiers such that replaying $[\![Q]\!]_{\mathcal{D}|_S} = [\![Q]\!]_{\mathcal{D}}$, where $\mathcal{D}|_S$ is $\mathcal{D}$ with all graphs not in S set to $\emptyset$ (including the default graph if $\texttt{DEFAULT} \notin S$). Intuitively, S identifies graphs that "witness" Q, analogous to why-provenance in databases [4]. This is not necessarily the *smallest* such set; it may be an overapproximation, particularly in the presence of $P_1 \ \texttt{OPT} \ P_2$ queries [24]. Alternative, more precise notions of source (for example involving triple-level annotations [8]) could also be used.

We define the provenance of an atomic update by translation to a sequence of updates that, in addition to performing the requested updates to a given named graph, also

constructs some auxiliary named graphs and triples (provenance record) in a special named graph for provenance information called $prov$ (the provenance graph). We apply this translation to each update posed by the user, and execute the resulting updates directly without further translation. We detail how provenance information should be attached to each SPARQL Update operation.

We consider simple forms of insert and delete operations that target a single, statically known, named graph g; full SPARQL Updates including simultaneous insert and delete operations can also be handled. In what follows, we write "(metadata)" as a placeholder where extra provenance metadata (e.g. time, author, etc. as in Dublin Core or further information given by the PROV vocabulary [15]) may be added. DROP commands simply end the provenance collection, but previous versions of the graph should still be available.

– A graph creation of a new graph `CREATE` g is translated to

$$
\begin{array}{l}
\texttt{CREATE}\ g; \\
\texttt{CREATE}\ g\_v_0; \\
\texttt{INSERT DATA}\ \{\texttt{GRAPH}\ prov\ \{ \\
\quad \langle g\ \texttt{version}\ g\_v_0\rangle, \langle g\ \texttt{current}\ g\_v_0\rangle, \\
\quad \langle u_1\ \texttt{type create}\rangle, \langle u_1\ \texttt{output}\ g\_v_0\rangle, \\
\quad \langle u_1\ \texttt{meta}\ m_i\rangle, \text{(metadata)} \\
\}\}
\end{array}
$$

– A drop operation (deleting a graph) `DROP` g is handled as follows, symmetrically to creation:

$$
\begin{array}{l}
\texttt{DROP}\ g; \\
\texttt{DELETE WHERE}\ \{\texttt{GRAPH}\ prov\ \{\langle g\ \texttt{current}\ g\_v_i\rangle\}\}; \\
\texttt{INSERT DATA}\ \{\texttt{GRAPH}\ prov\ \{ \\
\quad \langle u_i\ \texttt{type drop}\rangle, \langle u_i\ \texttt{input}\ g\_v_i\rangle, \\
\quad \langle u_i\ \texttt{meta}\ m_i\rangle, \text{(metadata)} \\
\}\}
\end{array}
$$

where g_v_i is the current version of g. Note that since this operation deletes g, after this step the URI g no longer names a graph in the store; it is possible to create a new graph named g, which will result in a new sequence of versions being created for it. The old chain of versions will still be linked to g via the `version` edges, but there will be a gap in the chain.

– A clear graph operation `CLEAR` g is handled as follows:

$$
\begin{array}{l}
\texttt{CLEAR}\ g; \\
\texttt{DELETE WHERE}\ \{\texttt{GRAPH}\ prov\ \{\langle g\ \texttt{current}\ g\_v_i\rangle\}\}; \\
\texttt{INSERT DATA}\ \{\texttt{GRAPH}\ prov\ \{ \\
\quad \langle g\ \texttt{version}\ g\_v_{i+1}\rangle, \langle g\ \texttt{current}\ g\_v_{i+1}\rangle, \\
\quad \langle u_i\ \texttt{type clear}\rangle, \langle u_i\ \texttt{input}\ g\_v_i\rangle, \\
\quad \langle u_i\ \texttt{output}\ g\_v_{i+1}\rangle, \langle u_i\ \texttt{meta}\ m_i\rangle, \\
\quad \text{(metadata)} \\
\}\}
\end{array}
$$

– A load graph operation LOAD h INTO g is handled as follows:

LOAD h INTO g;
DELETE WHERE {GRAPH $prov$ {$\langle g$ current $g\_v_i\rangle$}};
INSERT DATA {GRAPH $prov$ {
$\langle g$ version $g\_v_{i+1}\rangle$, $\langle g$ current $g\_v_{i+1}\rangle$,
$\langle u_i$ type load$\rangle$, $\langle u_i$ input $g\_v_i\rangle$,
$\langle u_i$ output $g\_v_{i+1}\rangle$, $\langle u_i$ source $h_j\rangle$,
$\langle u_i$ meta $m_i\rangle$, (metadata)
}}

where h_j is the current version of h. Note that a load will not create any new graphs because both the source and target should already exist. If no target exists, a new graph is created as outlined above with using the create operation.

– An insertion INSERT {GRAPH g {C}} WHERE P is translated to a sequence of updates that creates a new version and links it to URIs representing the update, as well as links to the source graphs identified by the query provenance semantics and a named graph containing the inserted triples:

CREATE g_u_i;
INSERT {GRAPH g_u_i {C}} WHERE P;
INSERT {GRAPH g {C}} WHERE P;
CREATE g_v_{i+1};
LOAD g INTO g_v_{i+1};
DELETE DATA {GRAPH $prov$ {$\langle g$ current $g\_v_i\rangle$}};
INSERT DATA {GRAPH $prov$ {
$\langle g$ version $g\_v_{i+1}\rangle$, $\langle g$ current $g\_v_{i+1}\rangle$,
$\langle u_i$ input $g\_v_i\rangle$, $\langle u_i$ output $g\_v_{i+1}\rangle$,
$\langle u_i$ type insert$\rangle$, $\langle u_i$ data $g\_u_i\rangle$
$\langle u_i$ source $s_1\rangle, \ldots, \langle u_i$ source $s_m\rangle$,
$\langle u_i$ meta $m_i\rangle$, (metadata)}}

where $s_1, \ldots, s_m$ are the source graph names of P.

– A deletion DELETE {GRAPH g {C}} WHERE P is handled similarly to an insert, except for the update type annotation.

CREATE g_u_i;
INSERT {GRAPH g_u_i {C}} WHERE P;
DELETE {GRAPH g {C}} WHERE P;
CREATE g_v_{i+1};
LOAD g INTO g_v_{i+1};
DELETE DATA {GRAPH $prov$ {$\langle g$ current $g\_v_i\rangle$}};
INSERT DATA {GRAPH $prov$ {
$\langle g$ version $g\_v_{i+1}\rangle$, $\langle g$ current $g\_v_{i+1}\rangle$,
$\langle u_i$ input $g\_v_i\rangle$, $\langle u_i$ output $g\_v_{i+1}\rangle$,
$\langle u_i$ type delete$\rangle$, $\langle u_i$ data $g\_u_i\rangle$
$\langle u_i$ source $s_1\rangle, \ldots, \langle u_i$ source $s_m\rangle$,
$\langle u_i$ meta $m_i\rangle$, (metadata)}}

Note that we still insert the deleted tuples into the g_u_i.

- The DELETE $\{C\}$ INSERT $\{C'\}$ WHERE P update can be handled as a delete followed by an insert, with the only difference being that both update steps are linked to the same metadata.
- The COPY h TO g, MOVE h TO g, and ADD h TO g commands can be handled similarly to LOAD h INTO g; the only subtlety is that if $g = h$ then these operations have no visible effect, but the provenance record should still show that these operations were performed.

Our approach makes a design decision to treat DELETE $\{C\}$ INSERT $\{C'\}$ WHERE P as a delete followed by an insert. In SPARQL Update, the effect of a combined delete–insert is not the same as doing the delete and insert separately, because both phases of a delete–insert are evaluated against the same data store before any changes are made. However, it is not clear that this distinction needs to be reflected in the provenance record; in particular, it is not needed to ensure correct reconstruction. Moreover, the connection between the delete and insert can be made explicit by linking both to the same metadata, as suggested above. Alternatively, the deletion and insertion can be treated as a single compound update, but this would collapse the distinction between the "sources" of the inserted and deleted data, which seems undesirable for use-cases such as version control.

Also note that our method does not formally take into account tracking the provenance of inferences. This is because of the complex interactions between SPARQL Update and the large number of possible (RDFS and the many varieties of OWL and OWL2) inference mechanisms and also because, unlike other research in the area [8], we do not consider it a requirement or even a desirable feature that inferences be preserved between updates. It is possible that an update will invalidate some inferences or that a new inference regime will be necessary. A simple solution would be that if the inferences produced by a reasoning procedure are necessary to be tracked with a particular graph, the triples resulting from this reasoning procedure should be materialized into the graph via an insert operation, with the history record's metadata specifying instead of a SPARQL Update statement the particular inference regime used. We also do not include a detailed treatment of blank nodes that takes their semantics as existential variables, as empirical research has in general shown that blank nodes are generally used as generic stable identifiers rather than existential variables, and thus can be treated as simply minting unique identifiers [11].

5 Update Provenance Vocabulary

For the provenance graph itself, we propose the following lightweight vocabulary called the "Update Provenance Vocabulary" (UPD) given in Table 2. Every time there is a change to a provenance-enabled graph by SPARQL Update, there is the addition of a provenance record to the provenance graph using the UPD vocabulary, including information such as an explicit time-stamp and the text of the SPARQL update itself. Every step in the transaction will have the option of recording metadata using W3C PROV vocabulary (or even some other provenance vocabulary like OPM) explicitly given by the "meta" link in our vocabulary and semantics, with UPD restricted to providing a record

of the 'cut-and-paste' operations needed for applications of dynamic provenance like version control. We align the UPD vocabulary as a specialization of the W3C PROV vocabulary. A graph (`upd:graph`) is a subtype of `prov:Entity` and an update of a graph (`upd:update`) is a subtype of `prov:Activity`. For inverse properties, we use the inverse names recommended by PROV-O [15].

Name	Description	PROV Subtype
upd:input	Link to provenance record from graph before an update	prov:wasUsedBy
upd:output	Link from provenance record to a graph after update	prov:generated
upd:data	Changed data in insert/delete operation	prov:wasUsedBy
upd:version	Sequential link forward in time between a version of a graph and an update	prov:hadRevision
upd:prevVersion	sequential link backwards in time between a version of a graph and an update	prov:wasRevisionOf
upd:type	Type of update operation (insert, delete, load, clear, create, or drop)	prov:type
upd:current	Link to most current state of graph	prov:hadRevision
upd:source	Any other graph that was consulted by the update	prov:wasUsedBy
upd:meta	Link to any metadata about the graph	rdfs:seeAlso
upd:user	User identifier (string or URI)	prov:wasAttributedTo
upd:text	Text of the SPARQL Update Query	prov:value
upd:time	Time of update to the graph	prov:atTime

Fig. 2. Lightweight Update Provenance Vocabulary (UPD)

6 Implementation Considerations

So far we have formalized a logical model of the provenance of a graph as it evolves over time (which allow us to derive its intermediate versions), but we have not detailed how to store or query the intermediate versions of a graph efficiently. For any given graph one should likely store the most up-to-date graph so that queries on the graph in its present state can be run without reconstructing the entire graph. One could to simply store the graph `G_vi` resulting from each update operation in addition to the provenance record, but this would lead to an explosive growth in storage requirements. This would also be the case even for the provenance graph if the storage of an auxiliary graph `G_ui` in a provenance record involved many triples, although we allow this in the UPD vocabulary as it may be useful for some applications. For those operating over large graphs, the contents of the named graphs `G_ui` that store inserted or deleted triples can be represented more efficiently by just storing the original graph and the SPARQL Update statements themselves in each provenance record given by the `upd:text` property, and not storing the auxiliary named graphs given by `upd:data`.

Strategically, one can trade computational expense for storage in provenance, due to the immutability of the provenance information. A hybrid methodology to ameliorate the cost of reconstruction of the version of a graph would be to store the graph at various

temporal intervals (i.e. "snapshots"). For small graphs where storage cost is low and processing cost is high, it would make more sense to store all provenance information for the entire graph. In situations where the cost of processing is high and storage cost is low, storing the SPARQL Updates and re-running them makes sense to reconstruct the graph. In this case, it also makes sense to store "snapshots" of the graph at various intervals to reduce processing cost. Simulation results for these scenarios are available.[3]

7 Conclusion

Provenance is a challenging problem for RDF. By extending SPARQL Update, we have provided a method to use W3C PROV (and other metadata vocabularies) to keep track of the changes to triple-stores. We formalized this approach by drawing on similar work in database archiving and copy-paste provenance, which allow us to use SPARQL Update provenance records to reconstruct graphs at arbitrary instances in time. This work is a first step in addressing the important issue of RDF version control. We hope this will contribute to discussion of how to standardize descriptions of changes to RDF datasets, and even provide a way to translate changes to underlying (e.g. relational or XML) databases to RDF representations, as the same underlying "cut-and-paste" model has already been well-explored in these kinds of databases [2]. Explorations to adapt this work to the Google Research-funded DatabaseWiki project, and implementation performance with real-world data-sets is a next step [3]. A number of areas for theoretical future work remain, including the subtle issue of combining it with RDFS inferences [8] or special-purpose SPARQL provenance queries [25,16].

Acknowledgements This work was supported in part by EU FP7 project DIACHRON (grant number 601043). The authors, their organizations and project funding partners are authorized to reproduce and distribute reprints and on-line copies for their purposes notwithstanding any copyright annotation hereon.

References

1. M. Arenas, C. Gutierrez, and J. Perez. On the semantics of SPARQL. In *Semantic Web Information Management: A Model Based Perspective*. Springer, 1st edition, 2009.
2. Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 539–550, New York, NY, USA, 2006. ACM.
3. Peter Buneman, James Cheney, Sam Lindley, and Heiko Müller. DBWiki: a structured wiki for curated data and collaborative data management. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1335–1338. ACM, 2011.
4. Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, number 1973 in LNCS, pages 316–330, 2001.
5. Peter Buneman and Egor Kostylev. Annotation algebras for RDFS. In *SWPM*, 2010.
6. Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs. *Web Semant.*, 3:247–267, December 2005.

[3] *http://www.ibiblio.org/hhalpin/homepage/notes/dbprov-implementation.pdf*

7. James Cheney. The semantics of the PROV data model. W3C Note, April 2013. http://www.w3.org/TR/2013/NOTE-prov-sem-20130430/.
8. Giorgos Flouris, Irini Fundulaki, Panagiotis Pediaditis, Yannis Theoharis, and Vassilis Christophides. Coloring RDF triples to capture provenance. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 196–212, Berlin, Heidelberg, 2009. Springer-Verlag.
9. Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40, New York, NY, USA, 2007. ACM.
10. Claudio Gutierrez, Carlos Hurtado, and Ro Vaisman. Temporal RDF. In *European Conference on the Semantic Web (ECSW'05)*, pages 93–107, 2005.
11. Harry Halpin. Is there anything worth finding on the semantic web? In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 1065–1066, New York, NY, USA, 2009. ACM.
12. Ross Horne, Vladimiro Sassone, and Nicholas Gibbins. Operational semantics for SPARQL Update. In *JIST*, pages 242–257, 2011.
13. Graham Klyne and Paul Groth. Provenance Access and Query. W3C Note, April 2013. http://www.w3.org/TR/2013/NOTE-prov-aq-20130430/.
14. Natalia Kwasnikowska, Luc Moreau, and Jan Van den Bussche. A formal account of the Open Provenance Model. December 2010.
15. Timothy Lebo, Satya Sahoo, and Deborah McGuinness. PROV-O: The PROV ontology. W3C Recommendation, April 2013. http://www.w3.org/TR/2013/REC-prov-o-20130430/.
16. Nuno Lopes, Axel Polleres, Umberto Straccia, and Antoine Zimmermann. AnQL: SPARQLing up annotated RDFS. In *Proceedings of the 9th International Semantic web Conference - Part I*, ISWC'10, pages 518–533, Berlin, Heidelberg, 2010. Springer-Verlag.
17. L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J Van den Bussche. The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743–756, June 2011.
18. Luc Moreau. Provenance-based reproducibility in the semantic web. *J. Web Sem.*, 9(2):202–221, 2011.
19. Luc Moreau and Paolo Missier. PROV-DM: The PROV data model. W3C Recommendation, August 2013. http://www.w3.org/TR/2013/REC-prov-dm-20130430/.
20. Vicky Papavassiliou, Giorgos Flouris, Irini Fundulaki, Dimitris Kotzinos, and Vassilis Christophides. On detecting high-level changes in RDF/S KBs. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 473–488, Berlin, Heidelberg, 2009. Springer-Verlag.
21. Jorge Perèz, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *Transactions on Database Systems*, 34(3):A16, 2009.
22. Simon Schenk, Paul Gearon, and Alexandre Passant. SPARQL 1.1 Update. W3C Recommendation, March 2013. http://www.w3.org/TR/2013/REC-sparql11-update-20130321/.
23. Richard Thomas Snodgrass. *Developing time-oriented database applications in SQL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.
24. Yannis Theoharis, Irini Fundulaki, Grigoris Karvounarakis, and Vassilis Christophides. On provenance of queries on semantic web data. *IEEE Internet Computing*, 15(1):31–39, 2011.
25. Octavian Udrea, Diego Reforgiato Recupero, and V. S. Subrahmanian. Annotated RDF. *ACM Trans. Comput. Logic*, 11:A10, January 2010.